UCS645 – Assignment 1

Name: Lavish Arora
Roll No: 102483041
Course: UCS645

Q1. Parallel DAXPY Using OpenMP

Methodology:
DAXPY (Double precision A·X Plus Y) is a fundamental vector operation widely used in scientific computing. Each iteration is independent, making it ideal for data-parallel execution using OpenMP. In this experiment, the loop iterations are divided among threads using 'parallel for'. Performance improves as threads increase until hardware limits are reached. Beyond this point, memory bandwidth saturation and thread management overhead reduce efficiency.

Resource Utilization:
The DAXPY operation mainly utilizes CPU cores and shared memory bandwidth. CPU utilization increases with more threads, but memory bandwidth becomes the limiting factor, causing reduced efficiency beyond the optimal thread count.

Aim:
To analyze speedup of DAXPY operation using OpenMP by varying thread counts.

Operation: X[i] = a * X[i] + Y[i]
Vector Size: 2^16

Objective: To study speedup by increasing number of threads.

Threads	Execution Time (seconds)
2	0.002374
4	0.000899
8	0.001511

```
PS C:\Users\asus\OneDrive\Desktop\UCS645\LAB1> clang daxpy.c -fopenmp -O2 -o daxpy.exe
PS C:\Users\asus\OneDrive\Desktop\UCS645\LAB1> $env:OMP_NUM_THREADS=2; .\daxpy.exe
Threads = 2
Time = 0.002374
PS C:\Users\asus\OneDrive\Desktop\UCS645\LAB1> $env:OMP_NUM_THREADS=4; .\daxpy.exe
Threads = 4
Time = 0.000899
PS C:\Users\asus\OneDrive\Desktop\UCS645\LAB1> $env:OMP_NUM_THREADS=8; .\daxpy.exe
Threads = 8
Time = 0.001511
PS C:\Users\asus\OneDrive\Desktop\UCS645\LAB1> $env:OMP_NUM_THREADS=8; .\daxpy.exe
Threads = 8
```

Analysis:
Execution time decreases when threads increase from 2 to 4 due to better parallel utilization. At 8 threads, time slightly increases because DAXPY is memory-bound and threads compete for memory bandwidth.

Conclusion:
Maximum speedup is achieved at 4 threads. Increasing threads beyond core count introduces overhead.

Aim:
To compare performance of 1D and 2D parallel matrix multiplication using OpenMP.

Methodology:
Matrix multiplication is a compute-intensive workload with three nested loops. Parallelization can be applied either to a single outer loop (1D) or to two nested loops (2D). In shared-memory systems, 1D parallelization already provides good workload distribution by assigning rows to threads. 2D parallelization increases the number of parallel tasks but also introduces higher scheduling and synchronization overhead.

Q2. Parallel Matrix Multiplication

Resource Utilization:
Matrix multiplication heavily utilizes CPU computation and cache memory. In 1D parallelization, CPU cores are efficiently used, while 2D parallelization increases scheduling overhead without significantly improving cache usage, leading to similar overall utilization.

Matrix Size: 500 x 500
Two implementations: 1D Parallelization and 2D Parallelization

Method	Execution Time (seconds)
1D Parallel	0.030038
2D Parallel	0.029894

```
PS C:\Users\asus\OneDrive\Desktop\UCS645\LAB1> $env:OMP_NUM_THREADS=4; .\matrix1d.exe
1D Time=0.030038
PS C:\Users\asus\OneDrive\Desktop\UCS645\LAB1> notepad matrix2d.c
PS C:\Users\asus\OneDrive\Desktop\UCS645\LAB1> clang matrix2d.c -fopenmp -O2 -o matrix2d.exe
PS C:\Users\asus\OneDrive\Desktop\UCS645\LAB1> $env:OMP_NUM_THREADS=4; .\matrix2d.exe
2D Time=0.029894
```

Analysis:
Both methods show nearly equal performance. 1D parallelization already distributes workload efficiently. 2D parallelization adds scheduling overhead with negligible improvement.

Aim:
To approximate π using OpenMP parallel numerical integration and study reduction.

Methodology:
The value of π is approximated using numerical integration (Riemann sum). The total interval is divided among threads, where each thread computes a partial sum. The OpenMP reduction clause is used to safely combine these partial results, avoiding race conditions. This approach demonstrates cooperative parallel computation and highlights the importance of reductions in shared-memory programming.

Conclusion:
1D parallelization is sufficient for this matrix size. 2D does not provide significant benefit.

Resource Utilization:
The π computation effectively utilizes CPU cores through parallel workload distribution. Memory usage is minimal, and the reduction operation ensures efficient aggregation of partial results with low synchronization overhead.

Q3. Parallel Computation of Pi

Pi is computed using numerical integration of:
$\pi = \int_0^1 4/(1+x^2)\,dx$
Reduction clause is used to avoid race condition.

Calculated Pi	Execution Time (seconds)
3.141593	0.028451

Analysis:
Parallel reduction combines partial sums safely. Execution time is significantly reduced while maintaining accuracy.

Conclusion:
OpenMP reduction enables efficient cooperative computation of Pi.

```
PS C:\Users\asus\OneDrive\Desktop\UCS645\LAB1> notepad pi.c
PS C:\Users\asus\OneDrive\Desktop\UCS645\LAB1> clang pi.c -fopenmp -O2 -o pi.exe
PS C:\Users\asus\OneDrive\Desktop\UCS645\LAB1> $env:OMP_NUM_THREADS=4; .\pi.exe
Pi=3.141593
Time=0.028451
```